Set Based Analysis of ML Programs

(Extended Abstract)

NEVIN HEINTZE

July 1993
CMU-CS-93-193

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also appears as Fox Memorandum CMU-CS-FOX-93-03

Abstract

Reasoning about a program by treating program variables as sets of "values" leads to a simple, accurate and intuitively appealing notion of program approximation. This paper presents such an approach for the compile-time analysis of ML programs. To develop the core ideas of the analysis, we consider a simple untyped call-by-value functional language. Starting with an operational semantics for the language, we develop an approximate "set based" operational semantics which formalizes the intuition of treating program variables as sets. The key result of the paper is an $O(n^3)$ algorithm for computing the set based approximation of a program. We then show how the analysis can be extended in a natural way to deal with arrays, arithmetic, exceptions and continuations. We also briefly describe results from an implementation of this analysis for ML programs.

This work was sponsored by the Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: program analysis, ML, set based analysis, set constraints, type theory, subtypes, binding time analysis, control flow analysis.

1 Introduction

The motivation of this paper is the development of a compile-time analysis of functional programs that combines the following properties:

- It provides relatively accurate information about a program, with particular emphasis on the program's data structures.
- The underlying notion of approximation has a simple uniform definition, and the results of the analysis are intuitive and predictable.
- It is flexible and easily modified to incorporate arithmetic and operations such as assignment, callcc and exception handling.
- It is practical.

To this end, we consider an approach to program analysis based solely on the idea of ignoring inter-variable dependencies. To illustrate what is meant by inter-variable dependencies, consider the following two ML programs.

```
let fun mk_list (u, v) = [u, v]
in
   mk_list(1,2);
   mk_list(3,4)
end
```

Program 1

```
let fun append(x :: xs, y) = x :: append(xs, y)
      | append(nil, y') = y'
   fun rev(z :: zs) = append(rev zs, [z])
      | rev nil = nil
in rev [1,2,3,4]
end
```

Program 2

During the execution of Program 1, the body of the function mk_list is executed in two environments: $[u \mapsto 1, v \mapsto 2]$ and $[u \mapsto 3, v \mapsto 4]$. Inter-variable dependencies arise here in the sense that the variable u takes the value 1 exactly when v takes 2, and u takes 3 when v takes 4. In general, we say that inter-variable dependencies arise whenever the set of environments encountered at some program point is such that fixing a value for one or more variables restricts the possible values of the other variables.

Such dependencies may be ignored by treating the program variables as denoting sets of values instead of individual values. In Program 1, the sets for u and v are {1, 3} and {2, 4} respectively. If program variables are treated as sets, then the result of Program 1 is approximated by the *set* of values {[1,2], [1,4], [3,2], [3,4]}, in contrast to its actual result which is the single value [3,4].

In Program 2, dependencies arise between the variables x, xs and y in the function append, and between z and zs in the function rev. If the values of variables are collected into sets, then we obtain the set {1,2,3,4} for both x and z. Using this information, a set based interpretation of the program can be developed as follows. Consider the definition of append. From the first clause, we see that the values returned by append include values $1 :: l$, $2 :: l$, $3 :: l$ and 4

:: l where l is some list returned by append. From the second clause, the values returned by append include any value of y, and noting the call append(rev zs, [z]) in the definition of rev, these values include the singleton lists [1], [2], [3] and [4]. Combining these two observations, it is easy to see that the set based interpretation of Program 2 yields the set of all lists constructed from 1, 2, 3 and 4.

The notion of set based approximation can be extended in a variety of ways. For example, consider programs involving arrays. In keeping with the methodology of ignoring dependencies, we shall ignore the dependencies between subscripts and array values. In essence, we treat an array as a set of values. When the array is updated, a new value is added to this set. When the array is subscripted, the whole set is returned. For example, the set based approximation of Program 3 yields the set of all values obtained by summing any number of 3's and 4's i.e. $\{3n + 4m : m \geq 0, n \geq 0, m + n \geq 1\}$.

```
let fun cum (arr : int array) =
    let  fun f 0 = arr sub 0
           | f i = (arr sub i) + f(i − 1)
    in
        f ((length arr) − 1)
    end
    val arr = array(10, 3)
in
    update(arr, 6, 4);
    cum arr
end
```

Program 3

```
fun map f (x :: l) = (f x) :: (map f l)
    | map f nil = nil

val t = [1,2,3]
val d = dynamic
val u = map (fn x ⇒ (x, d)) l
val v = map (fn (x, y) ⇒ x) u
val w = map (fn (x, y) ⇒ y) u
```

Program 4

Set based approximation can also be extended to deal with non-standard values. For example, to perform a binding time analysis [5, 13, 17], a non-standard value dynamic is introduced to represent a value that will not be known until "run-time". To illustrate this, consider Program 4. The set based approximation of this program yields the following information[1] about the variables u, v and w: u is a list of pairs whose first element is either 1, 2 or 3 and whose second argument is dynamic; v is a list of 1's, 2's and 3's, and w is a list of dynamic's.

To summarize, the analysis developed here is based on the notion of ignoring inter-variable dependencies by treating variables as sets. In other words, the environments encountered at each point in a program are collapsed into a single set environment (mapping from variables into sets). Strictly speaking, there are three kinds of dependencies that are ignored in set based analysis. First, dependencies between different variables are ignored – this was illustrated by Program 1. Second, dependencies between different occurrences of the same variable are ignored. For example the approximation of Program 5 yields {[1,1], [1,2], [2,1], [2,2]} and not {[1,1], [2,2]}. Third, dependencies between the domain and codomain of functions are ignored. For example the approximation of Program 6 yields {2,3} and not {3}.

[1] We note that this information is in fact obtained only if the analysis of map is "polyvariant" (that is, provided there can be different "versions" of map). We refer to this issue later in the paper.

```
let fun f x = [x,x]
in
   f 1;
   f 2;
end
```
Program 5

```
let fun g 1 = 2
      | g 2 = 3
in
   g 1;
   g 2;
end
```
Program 6

In this paper we do two things. First, we develop the underlying ideas of set based analysis. We give a simple and natural formalization of the notion of set based approximation, and then present an algorithm (based on constructed and solving set constraints) for computing this approximation. This is carried out in the context of a small untyped call-by-value functional language that is intended to be suggestive of a number of aspects of ML [15]. Second, we describe an implementation for the set based analysis of ML programs, which extends the basic notions of set based analysis to arithmetic, arrays, continuations and exceptions. This implementation is built on the LAMBDA intermediate representation of the SML/NJ compiler [3]. Typical execution times are from less than a second for small programs, to a couple of seconds for moderate sized programs of the order of 1000 lines[2]. The implementation subsumes and generalizes aspects of type analysis, safety analysis, control flow analysis, structure sharing and usage analysis, interval analysis, and binding time analysis. Applications of the implementation include improved code generation (the information obtained can be used to guide data-structure representation and in-lining, and remove redundant tests such as array bounds checking), partial evaluation and checking static program properties (for example, better checking of non-exhaustive pattern matching).

Related Literature

The idea of defining program approximation by treating program variables as set of values has been used previously in the analysis of logic programs and imperative programs by Jaffar and the present author [7, 8, 9]. In the context of functional programs, work on type inference by Mishra and Reddy [16] is similar in spirit, although in [16] substantial restrictions are placed on types (for example, they must be "tuple-distributive").

More closely related is work by Aiken and Wimmers [2], who extract type constraints from a program and provide a normalization procedure for solving these constraints over the domain of downward closed sets of finite elements (essentially the "ideal" model of types). However, the constraints used and their simplification algorithm are very different from those used in the present paper. Constraints have also been used in binding time analysis [11] and safety analysis [18]. In the former, the program approximation that arises is different from set based approximation (and in fact less accurate), but can be computed in almost-linear time. In the latter (which is based on closure analysis), the constraint are solved over subsets of a finite domain of "closures". In contrast, our constraints are solved over an infinite domain.

[2] It also runs all of the examples given in this section.

Perhaps the most closely related work is by Jones [12], where a grammar approach is presented to the analysis of lazy higher-order functional programs. The main aspect of our work that sets it apart from other works is that we start with a simple, intuitive definition of approximate semantics based on an operational semantics, and only then present algorithms (using constraints) that correspond exactly with this approximation. Moreover, we extend this analysis to deal with side effects and continuations in a uniform and intuitive manner.

2 Set Based Approximation

Consider a simple call-by-value functional language whose terms e are defined by

$$e ::= x \mid c(e_1, \ldots, e_n) \mid \lambda x.e \mid e_1\ e_2 \mid case(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3) \mid fix\ x.e$$

where x and y range over program variables and c ranges over a given set of (varying arity, "first-order") constants. It is convenient to adopt the usual convention that each bound variable is distinct. The operator *case* is essentially a very restricted form of the ML case expression and provides a mechanism for branching on the result of a computation as well as "deconstructing" values. The operator *fix* serves to express recursion[3].

The operational semantics for the evaluation of the closed terms of this language is given in Figure 1. The variables E and v range over environments and values respectively, and these are defined mutually recursively as follows. An *environment* E is a finite mapping from program variables into binding values. A *binding* value is either a value or an expression of the form $fix\ x.e$. A *value* v is of the form $c(v_1, \ldots, v_n)$ where the v_i are values, or a closure of the form $\langle E, \lambda x.e \rangle$ where E is an environment. If E is an environment then we write $dom(E)$ to denote the (finite) set of variables on which E is defined. The notation $E[x \mapsto exp]$ denotes the environment which maps x into exp and all other variables x' into $E(x')$. We write $\vdash e \rightarrow v$ if $E \vdash e \rightarrow v$ when E is the empty environment.

We now modify the operational semantics so that dependencies between variables are ignored. This is achieved by treating program variables as sets of values. To formalize this, first define that a *set environment* $\mathcal{E}$ is a finite mapping from variables into sets of binding values. We write $E \in \mathcal{E}$ to denote that $E(x) \in \mathcal{E}(x)$ for all $x \in dom(E)$. The *set based operational semantics*, presented in Figure 2, is essentially obtained by replacing environments E in the rules of Figure 1 by set environments[4] $\mathcal{E}'$. This replacement necessitates two kinds of changes to the rules. First, the two variable rules VAR-1 and VAR-2 are modified to accommodate the fact that $\mathcal{E}(x)$ is a set. Second, the rules that involve variable binding (APP, CASE-1, CASE-2 and FIX) are modified so that the binding information is dropped.

Observe that this second group of rules will, in general, lead to an unsound approximation.

[3] In $fix\ x.e$, the expression e shall typically be an abstraction.

[4] We remark that one reason for the explicit use of environments in the operational semantics in Figure 1 is precisely to enhance this intuition. However, the notion of set based approximation is not limited to this style of semantics. Analogous definitions can be made starting from an operational semantics that uses substitution.

$$E \vdash x \to E(x) \qquad (E(x) \neq \mathit{fix}\, y.e) \qquad \text{(VAR-1)}$$

$$\frac{E \vdash \mathit{fix}\, y.e \to v}{E \vdash x \to v} \qquad (E(x) = \mathit{fix}\, y.e) \qquad \text{(VAR-2)}$$

$$\frac{E \vdash e_1 \to \langle E', \lambda x.e\rangle \qquad E \vdash e_2 \to v' \qquad E'[x \mapsto v'] \vdash e \to v}{E \vdash e_1\, e_2 \to v} \qquad \text{(APP)}$$

$$\frac{E \vdash e_1 \to v_i,\ i = 1..n}{E \vdash c(e_1, \ldots, e_n) \to c(v_1, \ldots, v_n)} \qquad \text{(CONST)}$$

$$E \vdash \lambda x.e \to \langle E, \lambda x.e\rangle \qquad \text{(ABS)}$$

$$\frac{E \vdash e_1 \to c(v_1, \ldots, v_n) \qquad E[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash e_2 \to v}{E \vdash \mathit{case}(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3) \to v} \qquad \text{(CASE-1)}$$

$$\frac{E \vdash e_1 \to c'(v_1, \ldots, v_n) \qquad E[y \mapsto c'(v_1, \ldots, v_n)] \vdash e_3 \to v}{E \vdash \mathit{case}(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3) \to v} \quad (c \neq c') \quad \text{(CASE-2)}$$

$$\frac{E[x \mapsto \mathit{fix}\, x.e] \vdash e \to v}{E \vdash \mathit{fix}\, x.e \to v} \qquad \text{(FIX)}$$

Figure 1: Operational Semantics for the Simple Language.

$$\mathcal{E} \vdash x \rightsquigarrow v \qquad (v \in \mathcal{E}(x),\ v \neq \mathit{fix}\ y.e) \qquad \text{(VAR-1)}$$

$$\frac{\mathcal{E} \vdash \mathit{fix}\ y.e \rightsquigarrow v}{\mathcal{E} \vdash x \rightsquigarrow v} \quad (\mathit{fix}\ y.e \in \mathcal{E}(x)) \qquad \text{(VAR-2)}$$

$$\frac{\mathcal{E} \vdash e_1 \rightsquigarrow \langle E, \lambda x.e \rangle \qquad \mathcal{E} \vdash e_2 \rightsquigarrow v' \qquad \mathcal{E} \vdash e \rightsquigarrow v}{\mathcal{E} \vdash e_1\ e_2 \rightsquigarrow v} \qquad \text{(APP)}$$

$$\frac{\mathcal{E} \vdash e_1 \rightsquigarrow v_i,\ i = 1..n}{\mathcal{E} \vdash c(e_1, \ldots, e_n) \rightsquigarrow c(v_1, \ldots, v_n)} \qquad \text{(CONST)}$$

$$\mathcal{E} \vdash \lambda x.e \rightsquigarrow \langle E, \lambda x.e \rangle \qquad (E \in \mathcal{E}) \qquad \text{(ABS)}$$

$$\frac{\mathcal{E} \vdash e_1 \rightsquigarrow c(v_1, \ldots, v_n) \qquad \mathcal{E} \vdash e_2 \rightsquigarrow v}{\mathcal{E} \vdash \mathit{case}(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3) \rightsquigarrow v} \qquad \text{(CASE-1)}$$

$$\frac{\mathcal{E} \vdash e_1 \rightsquigarrow c'(v_1, \ldots, v_n) \qquad \mathcal{E} \vdash e_3 \rightsquigarrow v}{\mathcal{E} \vdash \mathit{case}(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3) \rightsquigarrow v} \quad (c \neq c') \qquad \text{(CASE-2)}$$

$$\frac{\mathcal{E} \vdash e \rightsquigarrow v}{\mathcal{E} \vdash \mathit{fix}\ x.e \rightsquigarrow v} \qquad \text{(FIX)}$$

Figure 2: Set Based Operational Semantics.

That is, certain set environments $\mathcal{E}$ will be such that for some closed terms e_0, $\vdash e_0 \rightarrow v$ but $\mathcal{E} \not\vdash e_0 \rightsquigarrow v$. We shall however always ensure that whenever one of these rules is applied, $\mathcal{E}$ is "sufficiently large" that it contains all bindings to variables. For rule APP, the required condition on $\mathcal{E}$ is $v' \in \mathcal{E}(x)$. Similar conditions can be given for the other rules involving binding. Note that it is not appropriate to just add these conditions as side conditions to the respective rules, since this would have the effect of reducing the number of derivations. Instead, we require that whenever one of the potentially unsafe rules is applied, the extra conditions are always met. To formalize this, define that $\mathcal{E}$ is *safe* with respect to a closed term e_0 if, for every derivation ending in $\mathcal{E} \vdash e_0 \rightsquigarrow v$, the following four conditions are met (we follow the notation established in Figure 2):

1. Every application of the rule APP is such that $v' \in \mathcal{E}(x)$.
2. Every application of the rule CASE-1 is such that $v_i \in \mathcal{E}(x_i), i = 1..n$.
3. Every application of the rule CASE-2 is such that $c'(v_1, \ldots, v_n) \in \mathcal{E}(y)$.
4. Every application of the rule FIX is such that $\mathit{fix}\ x.e \in \mathcal{E}(x)$.

Importantly, safety implies soundness in the following sense:

Theorem 1 (Soundness) *If $\mathcal{E}$ is safe wrt a closed term e_0, then $\vdash e_0 \rightarrow v$ implies $\mathcal{E} \vdash e_0 \rightsquigarrow v$.*

Proof: The proof follows by structural induction on the subderivations of the derivation $\vdash e \rightarrow v$. The induction hypothesis must be strengthened slightly to include a simple property about the closures that may be encountered. □

In essence, this proves that if we guess $\mathcal{E}$ so that it is safe, then the set based operational semantics provides a sound approximation of the execution of a term. However, given a term e_0, there are many correct choices for $\mathcal{E}$, and these give rise to different approximations of e_0. The following proposition implies that, given e_0, there is a canonical choice for $\mathcal{E}$, and that this choice gives rise to the most accurate approximation of e_0. First, define that the intersection of set environments $\mathcal{E}_1$ and $\mathcal{E}_2$, denoted $\mathcal{E}_1 \cap \mathcal{E}_2$, is given by: $(\mathcal{E}_1 \cap \mathcal{E}_2)(x) \stackrel{\text{def}}{=} \mathcal{E}_1(x) \cap \mathcal{E}_2(x)$, provided $\mathcal{E}_1(x)$ and $\mathcal{E}_2(x)$ are both defined. Then:

Proposition 1 (Minimality) *If $\mathcal{E}_1$ and $\mathcal{E}_2$ are safe wrt a closed term e_0, then so is $\mathcal{E}_1 \cap \mathcal{E}_2$. Moreover, $\mathcal{E}_1 \cap \mathcal{E}_2 \vdash e_0 \rightsquigarrow v$ implies $\mathcal{E}_1 \vdash e_0 \rightsquigarrow v$ and $\mathcal{E}_2 \vdash e_0 \rightsquigarrow v$.*

Proof: The proof here is straightforward and follows from the observation that any derivation $\mathcal{E}_1 \cap \mathcal{E}_2 \vdash e_0 \rightsquigarrow v$ can be replayed to give isomorphic derivations $\mathcal{E}_1 \vdash e_0 \rightsquigarrow v$ and $\mathcal{E}_2 \vdash e_0 \rightsquigarrow v$. □

This motivates the following definition[5].

[5] It is possible to give a direct definition of set based approximation, which avoids the minimization over $\mathcal{E}$. However such a definition is substantially more complex.

Definition 1 (Set Based Approximation) *Let e_0 be a closed term. Let $\mathcal{E}_{min}$ be the least set environment that is safe wrt e_0. The set based approximation of e_0, denoted $sba(e_0)$, is defined by:*

$$sba(e_0) \stackrel{\text{def}}{=} \{v : \mathcal{E}_{min} \vdash e_0 \rightsquigarrow v\} \quad \square$$

To summarize, the set based operational semantics approximates the execution of a term by collapsing all environments into one single set environment. No other form of approximation is employed. In particular, no use is made of abstract domains (such as those commonly employed in abstract-interpretation styles of program analysis [6]). We remark that the results of the analysis are typically infinite sets of values, and that we make no *a priori* requirement that these sets be finitely presentable.

3 Main Result

We now present the main result of the paper, which is an algorithm for computing $sba(e_0)$ for any closed term e_0. The structure of the algorithm is as follows. First, we construct *set constraints* corresponding to the input term e_0. In essence, these constraints express relationships between sets of values in such a way that a model of the constraints corresponds to the set based execution of e_0 in some safe set environment $\mathcal{E}$. Importantly, the least model of these constraints corresponds to execution in the smallest safe set environment, and hence to $sba(e_0)$. The second part of the algorithm is a simplification procedure for set constraints. In essence, this algorithm constructs an explicit representation of the least model of the input set constraints. This representation is in the form of a regular tree grammar. Note that no assumptions have been made about the adequacy of regular tree grammars. The fact that the least model of the set constraints (and hence $sba(e_0)$) can be represented using regular tree grammars is a by-product of the correctness proof of the algorithm.

Before describing the form of the set constraints employed by the algorithm, we first note that the environment part of closures in $sba(e_0)$ is essentially redundant. In particular, if $\mathcal{E}$ is the least set environment that is safe with respect to a closed term e_0, and if $sba(e_0)$ contains a closure $\langle E, \lambda x.e \rangle$, then $sba(e_0)$ must in fact contain all closures of the form $\langle E', \lambda x.e \rangle$ such that $E' \in \mathcal{E}$. This is because the set based operational semantics collapses all environments into the single set environment $\mathcal{E}$, and moreover, the only closures generated during the set based execution are via the (ABS) rule. In the computation of $sba(e_0)$, it is convenient to drop the redundant environment information in closures[6]. More formally, define an operator $\|v\|$ on values v, which forgets the environment part of closures, as follows:

$$\|v\| = \begin{cases} c & \text{if } v \text{ is the constant } c \\ \|v_1\| \; \|v_2\| & \text{if } v \text{ is } v_1 \; v_2 \\ \lambda x.e & \text{if } v \text{ is } \langle E, \lambda x.e \rangle \end{cases}$$

[6] We note that this can be recovered if needed, although it is not completely trivial to do so since values and environments are mutually dependent.

The algorithm presented in this section computes a representation (using regular tree grammars) of the set $\|sba(e_0)\| = \{\|v\| : v \in sba(e_0)\}$.

Set Constraints

The use of set constraints for analysis of programs dates back to the early works by Reynolds [19], and Jones and Muchnick [14], which employ constraints involving projection. The calculus of set constraints was first defined and studied in a general setting by Jaffar and the present author [9]. [9] also contained a decision procedure for a class of set constraints involving projection and intersection. Later works have provided algorithms for different classes of set constraints (Aiken and Murphy [1] have dealt with complementation and projection; Jaffar and the present author have dealt with set constraint operators that are designed for analyzing logic programs and imperative programs [7, 8], and combinations of set constraint techniques and abstract interpretation techniques [10]), as well as providing alternative proofs of previous results (Bachmair, Ganzinger and Waldmann [4] establish a connection between certain kinds of set constraints and a fragment of logic shown decidable by Löwenheim, and in the process provide alternative proofs of the earlier results in [1] and [9]).

We extend the basic set constraint calculus of [9] by adding operations to model function application and case statements. The form and meaning of these constraints is defined in the context of some given closed term e_0. We assume a fixed infinite class of *set variables*; set variables shall be denoted $\mathcal{W}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}$. We distinguish two special disjoint subclasses of set variables. First, for each program variable x in e_0, there is a distinct set variable $\mathcal{X}_x$ which shall be used to capture all of the values for the program variable x. Second, for each abstraction $\lambda x.e$ appearing in e_0, there is a distinct set variable $ran(\lambda x.e)$, the "range" of $\lambda x.e$, which shall be used to capture all of the values returned by applications of $\lambda x.e$ during execution. Now, in the context of the given term e_0, we define that a *set expression* (se) is either a set variable, an abstraction $\lambda x.e$ that appears in e_0, or of one of the forms $c(se_1, \ldots, se_2)$, $apply(se_1, se_2)$, $case(se_1, c(\mathcal{X}_1, \ldots, \mathcal{X}_n) \Rightarrow se_2, \mathcal{Y} \Rightarrow se_3)$ or $ifnonempty(se_1, se_2)$ (which shall be used later). The first form is used to model execution of expressions $c(e_1, \ldots, e_n)$, the second form models application, the third is for case statements, and the last is used to reason about emptiness. A *set constraint* is an expression of the form $\mathcal{X} \supseteq se$, and a *conjunction* C of set constraints is a finite collection of set constraints.

We now define the meaning of the set constraints. In essence, set expressions shall be interpreted as sets of values with the environment component of closures removed. Specifically, a set constraint *value* (sc-value) is either an abstraction $\lambda x.e$ that appears in e_0, or of the form $c(v_1, \ldots, v_n)$ where each v_i is an sc-value. An *interpretation* is a mapping from each set variable into a set of sc-values. Such an interpretation is extended to map set expressions to sets of sc-values as follows:

1. $\mathcal{I}(c(se_1, \ldots, se_n)) = \{c(v_1, \ldots, v_n) : v_i \in \mathcal{I}(se_i), i = 1..n\}$;
2. $\mathcal{I}(\lambda x.e) = \{\lambda x.e\}$;

3. $\mathcal{I}(ifnonempty(se_1, se_2)) = \text{if } \mathcal{I}(se_1) = \{\} \text{ then } \{\} \text{ else } \mathcal{I}(se_2)$;

4. $\mathcal{I}(apply(se_1, se_2)) = \left\{ v : \ \lambda x.e \in \mathcal{I}(se_1) \ \wedge \ \mathcal{I}(se_2) \neq \{\} \ \wedge \ v \in \mathcal{I}(ran(\lambda x.e)) \right\}$
provided $\lambda x.e \in \mathcal{I}(se_1)$ implies $\mathcal{I}(se_2) \subseteq \mathcal{I}(\mathcal{X}_x)$

5. $\mathcal{I}(case(se_1, c(\mathcal{X}_1, \ldots, \mathcal{X}_n) \Rightarrow se_2, \mathcal{Y} \Rightarrow se_3)) = S_1 \cup S_2$,
where (i) $S_1 = \{v : v \in \mathcal{I}(se_2) \ \wedge \ \exists c(v_1, \ldots, v_n) \in \mathcal{I}(se_1)\}$
(ii) $S_2 = \{v : v \in \mathcal{I}(se_3) \ \wedge \ \exists c'(v_1, \ldots, v_n) \in \mathcal{I}(se_1) \ s.t. \ c' \neq c\}$
(iii) $c(v_1, \ldots, v_n) \in \mathcal{I}(se_1)$ implies $v_i \in \mathcal{I}(\mathcal{X}_i)$, $i = 1..n$
(iv) $c'(v_1, \ldots, v_n) \in \mathcal{I}(se_1)$ where $c' \neq c$ implies $c'(v_1, \ldots, v_n) \in \mathcal{I}(\mathcal{Y})$

Note that the above interpretation of set expressions is somewhat unusual, because in parts 4 and 5 of the definition, the set expressions themselves impose restrictions on $\mathcal{I}$. If these conditions are not met, then the interpretation of the expression is undefined. An interpretation $\mathcal{I}$ is a *model* of a conjunction of constraints C if, for each constraint $\mathcal{X} \supseteq se$, it is the case that $\mathcal{I}(se)$ is defined and $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$. It is easy to verify a model intersection property for the set constraints used in this paper, and it follows that a conjunction C of constraints possesses a least model, denoted $lm(C)$, where models are ordered as follows: $\mathcal{I}_1 \supseteq \mathcal{I}_2$ if $\mathcal{I}_1(\mathcal{X}) \supseteq \mathcal{I}_2(\mathcal{X})$, for all set variables $\mathcal{X}$.

Constructing Set Constraints

The construction of set constraints from a term is described in Figure 3. (Strictly speaking, this is a somewhat simplified version – the complete version appears in Appendix I.) In the rules (APP), (CONST), (ABS) and (CASE), the variable $\mathcal{Y}$ is intended to be a new set variable that is not used in any other part of the derivation. Using these rules, we define

Definition 2 *Let e_0 be a closed term, then $SC(e_0)$ is the pair $(\mathcal{X}, C)$ such that $\triangleright e_0 : (\mathcal{X}, C)$.* □

To illustrate the construction of the constraints, consider the term $e_0 = e_1 \ e_2$ where e_1 is $\lambda f.c(f\ a, f\ b)$, e_2 is $\lambda x.x$ and a, b and c are constants[7]. For this term, we derive $\triangleright e_0 : (\mathcal{X}_1, C)$ where C consists of the constraints

$$\begin{array}{lll} \mathcal{X}_1 \supseteq apply(\mathcal{X}_2, \mathcal{X}_3) & \mathcal{X}_4 \supseteq apply(\mathcal{X}_f, a) & ran(e_1) \supseteq c(\mathcal{X}_4, \mathcal{X}_5) \\ \mathcal{X}_2 \supseteq e_1 & \mathcal{X}_5 \supseteq apply(\mathcal{X}_f, b) & ran(e_2) \supseteq \mathcal{X}_x \\ \mathcal{X}_3 \supseteq e_2 & & \end{array}$$

In $lm(C)$, $\mathcal{X}_1 = ran(e_1) = \{c(a,b), c(b,a), c(a,a), c(b,b)\}$, $\mathcal{X}_2 = \{e_1\}$, $\mathcal{X}_3 = \{e_2\}$, $\mathcal{X}_4 = \mathcal{X}_5 = ran(e_2) = \mathcal{X}_x = \{a, b\}$

[7] We shall write a as an abbreviation of $a()$.

$$\triangleright x : (\mathcal{X}_x, \{\}) \qquad \text{(VAR)}$$

$$\frac{\triangleright e_1 : (\mathcal{X}_1, C_1) \qquad \triangleright e_2 : (\mathcal{X}_2, C_2)}{\triangleright e_1\ e_2 : (\mathcal{Y}, \{\mathcal{Y} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2)\} \cup C_1 \cup C_2)} \qquad \text{(APP)}$$

$$\frac{\triangleright e_i : (\mathcal{X}_i, C_i),\ i = 1 \ldots n}{\triangleright c(e_1, \ldots, e_n) : (\mathcal{Y}, \{\mathcal{Y} \supseteq c(\mathcal{X}_1, \ldots, \mathcal{X}_n)\} \cup C_1 \cup \ldots \cup C_n)} \qquad \text{(CONST)}$$

$$\frac{\triangleright e : (\mathcal{X}, C)}{\triangleright \lambda x.e : (\mathcal{Y}, \{\mathcal{Y} \supseteq \lambda x.e, ran(\lambda x.e) \supseteq \mathcal{X}\} \cup C)} \qquad \text{(ABS)}$$

$$\frac{\triangleright e_1 : (\mathcal{Z}_1, C_1) \qquad \triangleright e_2 : (\mathcal{Z}_2, C_2) \qquad \triangleright e_3 : (\mathcal{Z}_3, C_3)}{\triangleright case(e_1, c(x_1, \ldots, x_n) \Rightarrow e_2, y \Rightarrow e_3) : (\mathcal{Y}, C \cup C_1 \cup C_2 \cup C_3)} \qquad \text{(CASE)}$$

$$\text{where } C = \{\mathcal{Y} \supseteq case(\mathcal{Z}_1, c(\mathcal{X}_{x_1}, \ldots, \mathcal{X}_{x_n}) \Rightarrow \mathcal{Z}_2, \mathcal{X}_y \Rightarrow \mathcal{Z}_3)\}$$

$$\frac{\triangleright e : (\mathcal{X}, C)}{\triangleright fix\ x.e : (\mathcal{X}_x, \{\mathcal{X}_x \supseteq \mathcal{X}\} \cup C)} \qquad \text{(FIX)}$$

Figure 3: Construction of Set Constraints (simplified version)

For presentational simplicity, the constraint construction given in Figure 3 is not in complete correspondence with $sba(e_0)$. To see this, consider the term $e_0 = e_1\ e_2$ where e_1 is $\lambda f.((\lambda u.f\ a)(\lambda w.f\ b))$ and e_2 is $\lambda x.x$. The least $\mathcal{E}$ that is safe with respect to e_0 maps f into $\{\lambda x.x\}$, u into $\{\lambda w.f\ b\}$ and x into $\{a\}$, and $sba(e_0)$ is $\{a\}$. However, the set constraint construction procedure traverses *all* subexpression of e_0. Hence $\mathcal{SC}(e_0)$ contains the set expressions $apply(\mathcal{X}_f, a)$ and $apply(\mathcal{X}_f, b)$. As a result, $\mathcal{X}_x$ must contain both a and b, and so the execution of e_0 is approximated by $\{a, b\}$. The problem is that the term $\lambda w.f\ b$ is never "executed" under the set based semantics, but is traversed by the set constraint construction process. To rectify this situation, the constraint construction must be such that if $\lambda x.e$ appears in e_0, then the constraints constructed for e are vacuously satisfied whenever $\mathcal{X}_x$ (the set of values for x) is empty. The complete constraint construction procedure appears in Appendix I. The correspondence between $sba(e_0)$ and $\mathcal{SC}(e_0)$ is given by the following Lemma[8]:

Lemma 1 *Let e_0 be a closed term, let $\mathcal{SC}(e_0)$ be $(\mathcal{X}, C)$ and let $\mathcal{I}_{lm} = lm(C)$. Then $\mathcal{I}_{lm}(\mathcal{X}) = \|sba(e_0)\|$.*

Proof Sketch: The proof is fairly lengthly and consists of two main parts. The first part involves modifying the definition of $\mathcal{E} \vdash e \rightarrow v$ so that environments are removed from closures. Call this new system $\vdash'$. The proof for this part involves showing a correspondence between $\vdash$ and $\vdash'$. The second part then relates $\vdash'$ with $\mathcal{SC}(e_0)$ by showing two relationships: (a) if $\mathcal{E}$ is the least set environment that is safe wrt e_0 (in $\vdash'$), then $\mathcal{E}$ can be used to define a model $\mathcal{I}$ of C such that $\mathcal{E} \vdash' e \rightsquigarrow v$ iff $v \in \mathcal{I}(\mathcal{X})$; and (b) if $\mathcal{I}$ is a model of C then we can define an $\mathcal{E}$ that is safe wrt e_0 such that if $\mathcal{E} \vdash' e \rightsquigarrow v$ then $v \in \mathcal{I}(\mathcal{X})$. In essence, part (a) shows that $\mathcal{I}_{lm}(\mathcal{X}) \subseteq \|sba(e_0)\|$, and part (b) shows that $\mathcal{I}_{lm}(\mathcal{X}) \supseteq \|sba(e_0)\|$. □

[8]We note that Lemma 1 holds using the constraint construction process described in Figure 3 if the following condition is satisfied: the least set environment $\mathcal{E}$ that is safe wrt e_0 is such that $\mathcal{E}(x) \neq \{\}$ for all x.

input a collection $\mathcal{C}$ of set constraints;
repeat
  if $\mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2)$ and $\mathcal{X}_1 \supseteq \lambda x.e$ both appear in $\mathcal{C}$ **then**
    add $\mathcal{X} \supseteq ran(\lambda x.e)$ to $\mathcal{C}$;
    add $\mathcal{X}_x \supseteq \mathcal{X}_2$ to $\mathcal{C}$;
  if $\mathcal{X} \supseteq case(\mathcal{Y}_1, c(\mathcal{W}_1, \ldots, \mathcal{W}_n) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3)$
      and $\mathcal{Y}_1 \supseteq c(\mathcal{Z}_1, \ldots, \mathcal{Z}_n)$ both appear in $\mathcal{C}$
      and $lm(explicit(\mathcal{C}))(\mathcal{Z}_i) \neq \{\}, i = 1..n$, **then**
    add $\mathcal{X} \supseteq \mathcal{Y}_2$ to $\mathcal{C}$;
    add $\mathcal{W}_i \supseteq \mathcal{Z}_i$ to $\mathcal{C}, i = 1..n$;
  if $\mathcal{X} \supseteq case(\mathcal{Y}_1, c(\mathcal{W}_1, \ldots, \mathcal{W}_n) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3)$
      and $\mathcal{Y}_1 \supseteq c'(\mathcal{Z}_1, \ldots, \mathcal{Z}_n)$ both appear in $\mathcal{C}$, where $c' \neq c$,
      and $lm(explicit(\mathcal{C}))(\mathcal{Z}_i) \neq \{\}, i = 1..n$, **then**
    add $\mathcal{X} \supseteq \mathcal{Y}_3$ to $\mathcal{C}$;
    add $\mathcal{W} \supseteq c'(\mathcal{Z}_1, \ldots, \mathcal{Z}_n)$ to $\mathcal{C}$;
  if $\mathcal{X} \supseteq ifnonempty(\mathcal{Y}_1, \mathcal{Y}_2)$ appears in $\mathcal{C}$ and $lm(explicit(\mathcal{C}))(\mathcal{Y}_1) \neq \{\}$ **then**
    add $\mathcal{X} \supseteq \mathcal{Y}_2$ to $\mathcal{C}$;
  if $\mathcal{X} \supseteq \mathcal{X}'$ and $\mathcal{X}' \supseteq ae$ both appear in $\mathcal{C}$,
      where ae is atomic and not a set variable, **then**
    add $\mathcal{X} \supseteq ae$ to $\mathcal{C}$;
until no step changes $\mathcal{C}$;
output $explicit(\mathcal{C})$;

Figure 4: Set Constraint Simplification Algorithm

Set Constraint Algorithm

We first address the issue of the output format of the algorithm. What we desire is an explicit representation of the least model of the set constraints, and specifically, of $sba(e_0)$. Since these sets are typically infinite, we must deal with finite representations of infinite sets. What is needed is a representation from which simple questions such as membership, emptiness and containment can be directly determined. The representation we use is based on a restricted form of set constraints. Specifically, define that a set expression is *atomic* if it is either an abstraction $\lambda x.e$ that appears in e_0, a set variable, or of the form $c(ae_1, \ldots, ae_n)$ where each ae_i is atomic. A constraint is in *explicit form* if it has the form $\mathcal{X} \supseteq ae$ where ae is an atomic set expression that is not a set variable (ae may of course *contain* set variables). A collection of constraints is in explicit form if each constraint therein is in explicit form. If $\mathcal{C}$ is a collection of constraints, then $explicit(\mathcal{C})$ denotes the explicit form constraints of $\mathcal{C}$. We note that explicit form constraints can be regarded as regular tree grammars by treating set variables as non-terminals and regarding a constraint $\mathcal{X} \supseteq ae$ as a production $\mathcal{X} \Rightarrow ae$.

The simplification algorithm accepts as input a collection of constraints (such as those constructed for a closed term e_0) and outputs an explicit form collection of constraints that has the same least model as the input collection. The main part of the algorithm involves exhaustively applying a series of simplification steps, and this serves to add new explicit form

constraints so that information about $lm(C)$ is incrementally transferred into the explicit part of C. The algorithm terminates exactly when all information about $lm(C)$ is present in $explicit(C)$. The details of the algorithm appear in Figure 4. The phrase "add $\mathcal{X} \supseteq se$ to C" is used to mean "add the constraint $\mathcal{X} \supseteq se$ if it does not already appear". An expression of the form $lm(explicit(C))(\mathcal{Y}) \neq \{\}$ indicates a test which can be performed as follows: construct $explicit(C)$, and (using standard algorithms), check to see if $\mathcal{Y}$ is empty in the least model of $explicit(C)$ (analogous procedures can be found in [7, 9]).

We note that the correctness of the algorithm relies on the fact that there are no "nested" set expressions. In other words, if an expression of the form $apply(se_1, se_2)$ appears in the constraints, then se_1 and se_2 are both set variables, and similarly for expressions involving *ifnonempty* and *case*. It is easy to see that $SC(e_0)$ satisfies this property, and it is trivial to verify that the algorithm preserves this property. The next lemma establishes the correctness of the simplification algorithm, and, combined with Lemma 1, proves Theorem 2.

Lemma 2 (Correctness of Algorithm) *The algorithm terminates on input C and outputs explicit form constraints C' such that $lm(C') = lm(C)$.*

Proof Sketch: Termination is straightforward to verify since the algorithm adds only constraints of the form $\mathcal{X} \supseteq ae$ where both $\mathcal{X}$ and ae are expressions that already appear in the constraints. The main part of the proof is to establish that the transformation steps are complete in the sense that when no further transformation steps can be applied, then $lm(C) = lm(explicit(C))$. This is achieved by showing that when no further transformation can be applied, the interpretation $lm(explicit(C))$ is in fact a model of C. □

Theorem 2 *Given a closed term e_0, there is an $O(n^3)$ algorithm to compute an explicit representation (which is equivalent to a regular tree grammar) of $||sba(e_0)||$.*

Proof: Let $SC(e_0)$ be $(\mathcal{X}, C)$. By Lemma 1, $lm(C)$ maps $\mathcal{X}$ into $||sba(e_0)||$. By Lemma 2, the set constraint simplification algorithm produces collection of constraints C' in explicit form when input with C. Moreover, $lm(C') = lm(C)$. Hence $lm(C')(\mathcal{X}) = lm(C)(\mathcal{X}) = ||sba(e_0)||$, and so C' provides an explicit representation of $||sba(e_0)||$. The $O(n^3)$ bound can be established as follows. First, the construction of constraints is linear in the size of e_0. Second, at most n^2 new constraints can be added by the simplification algorithm, and the cost of "adding" each new constraint (i.e. determining what other new constraints need to be added, given this constraint is added) can be bounded by $0(n)$. □

In addition, the algorithm trivially has an $O(n^2)$ space bound. We remark that this algorithm not only provides a way to compute $sba(e_0)$, but it also computes the least set environment that is safe wrt e_0.

4 Arrays, Continuations, Exceptions and Arithmetic

Thus far we have presented a formal development of the core ideas of set based analysis. We now informally outline the extensions we have employed for dealing with arrays, exceptions

and continuations. As outlined in the introduction, the set based treatment of arrays ignores dependencies between subscripts and values. That is, an array is treated as a set of values such that when the array is updated the new value(s) are added to this set, and when the array is accessed the set of values is returned. More concretely, for each place in the program where an array can be generated, we introduce a special distinct constant ar with two associated set variables $length(ar)$ and $contents(ar)$. We also introduce two new set expressions, $contentsof(se)$ and $update(se_1, se_2)$. In essence, the first denotes the union of the sets $contents(ar)$ such that ar is an element of se. The second is either (i) the empty set if either se_1 and se_2 is empty, (ii) the singleton set containing the unit value provided se_1 and se_2 are non-empty and $contents(ar) \supseteq se_2$ for all ar in se_1, or (iii) is undefined otherwise. The following rules are suggestive[9] of how constraints are constructed for programs involving arrays. In the first rule, ar is a new constant.

$$\frac{\triangleright e_1 : (\mathcal{X}_1, C_1) \qquad \triangleright e_2 : (\mathcal{X}_2, C_2)}{\triangleright array(e_1, e_2) : (\mathcal{Y}, \{\mathcal{Y} \supseteq ar, contents(ar) \supseteq se_1, length(ar) \supseteq se_2\} \cup C_1 \cup C_2)} \quad \text{(ARRAY)}$$

$$\frac{\triangleright e_1 : (\mathcal{X}_1, C_1) \qquad \triangleright e_2 : (\mathcal{X}_2, C_2)}{\triangleright e_1 \; sub \; e_2 : (\mathcal{Y}, \{\mathcal{Y} \supseteq contentsof(\mathcal{X}_1)\} \cup C_1 \cup C_2)} \quad \text{(SUBSCRIPT)}$$

$$\frac{\triangleright e_i : (\mathcal{X}_i, C_i),\ i = 1..3}{\triangleright update(e_1, e_2, e_3) : (\mathcal{Y}, \{\mathcal{Y} \supseteq update(\mathcal{X}_1, \mathcal{X}_2)\} \cup C_1 \cup C_2)} \quad \text{(UPDATE)}$$

Continuations are also modeled by introducing a new constant *cont* for each *callcc* appearing in a program. Each new constant has an associated set variable *contents(cont)*. In essence, this records the values that are thrown to the continuation. In effect, the constant *cont* passes into the term e a reference to the program point at which the *callcc* occurred (in fact it passes down the set variable corresponding to this point). The set expression $throw(se_1, se_2)$ is either (i) the empty set provided that $contents(cont) \supseteq se_2$ for each *cont* in se_1, or (ii) undefined otherwise.

$$\frac{\triangleright e : (\mathcal{X}, C)}{\triangleright callcc\, x.e : (\mathcal{Y}, \{\mathcal{Y} \supseteq \mathcal{X}, \mathcal{Y} \supseteq contents(cont), \mathcal{X}_x \supseteq cont\} \cup C)} \quad \text{(CALLCC)}$$

$$\frac{\triangleright e_1 : (\mathcal{X}_1, C_1) \qquad \triangleright e_2 : (\mathcal{X}_2, C_2)}{\triangleright throw(e_1, e_2) : (\mathcal{Y}, \{\mathcal{Y} \supseteq throw(\mathcal{X}_1, \mathcal{X}_2)\} \cup C_1 \cup C_2)} \quad \text{(THROW)}$$

Exceptions are modeled by introducing a distinct new set variable $\mathcal{EXC}$ to capture all of the exceptions that are raised during program execution. We note exceptions could be more accurately treated by introducing a new exception variable for each expression. This would provide better "separation" of the exceptions raised by different parts of a program, but at the cost of introducing more constraints. We are currently investigating this tradeoff.

[9] In particular, they are a simplification of the actual rules in the sense that Figure 3 simplifies Figure 5.

$$\frac{\triangleright e : (\mathcal{X}, C)}{\triangleright \textit{raise}\, e : (\mathcal{Y}, \{\mathcal{E}\mathcal{X}C \supseteq \mathcal{X}\} \cup C)} \quad \text{(RAISE)}$$

$$\frac{\triangleright e_1 : (\mathcal{X}_1, C_1) \qquad \triangleright e_2 : (\mathcal{X}_2, C_2)}{\triangleright e_1 \textit{ handle } (\lambda x.e_2) : (\mathcal{Y}, \{\mathcal{Y} \supseteq \mathcal{X}_1, \mathcal{Y} \supseteq \mathcal{X}_2, \mathcal{X}_x \supseteq \mathcal{E}\mathcal{X}C\} \cup C_1 \cup C_2)} \quad \text{(HANDLE)}$$

We conclude by considering arithmetic. In essence, we shall treat arithmetic operations like data constructors. For example, the analysis of the power program given below left yields the explicit form constraints given below right (where $\mathcal{X}$ is the set variable corresponding to the result of the program, and only solved form constraints relevant to $\mathcal{X}$ are shown).

```
let fun power(0, n) = 1
      | power(m, n) = n × power(m-1, n)
in
   power(3, 4)
end
```

$$\mathcal{X} \supseteq 4 \times \mathcal{X}$$
$$\mathcal{X} \supseteq 1$$

In other words, what is obtained is a description of *how* the value(s) in question were computed. The result for this program should be read as: the value computed by the program is either 1 or the result of multiplying 1 by 4 an arbitrary number of times. We omit further details for space reasons.

5 Implementation

An implementation of set based analysis for ML has been developed over the last two years. The system is build on top of the SML-NJ compiler. Starting with the LAMBDA intermediate representation of a program, our system incrementally builds and solves corresponding set constraints. Many of the set constraints that are generated are trivial, and so an important part of the effort to make the analyzer efficient was directed at ensuring that such constraints are solved "on-the-fly", and are never explicitly generated.

Space does not permit us to describe the treatment of arithmetic, and in particular, the treatment of if statements involving arithmetic expressions and comparisons. However, we note that early results suggest that the current implementation deals adequately with these constructs, and can be usefully employed to address issues such as the removal of unnecessary array bounds checking.

An important aspect of the implementation is "poly-variance" (the analysis analogue of polymorphism). That is, the implementation provides a mechanism to construct different "versions" of functions. In essence this is done by constraint duplication. However, for efficiency reasons, we wish to avoid multiple passes over the input LAMBDA expression, and instead we

first convert the LAMBDA expression into a compact internal format, from which multiple copies of constraints can be rapidly generated. A key aspect of polyvariance is how to control the generation of different versions of functions. One approach is to use the type information of a program (e.g. if a function is polymorphic, then it is likely to be useful to treat it as a poly-variant function). However, a goal of our implementation was to provide a generic analysis tool for functional programs, and so we did not want to commit to a typed language. Instead we chose a scheme in which the program is analyzed twice – the first pass is a "mono-variant" analysis, and the second pass uses information from the first to control a poly-variant analysis.

The following table presents some preliminary empirics for the implementation. We use three programs. The first program is the intmap structure from the SML-NJ compiler, which implements a mapping from integers to integers. The second models the game life, and is written in an applicative (rather than imperative) style. The third is the lexer generator from the ml-lex/ml-yacc collection. All times are in seconds on an PMAX 5000/200 with 64M and running Mach. The second column of the table gives the number of "equations" generated[10] (this excludes constraints that are solved on-the-fly). The third column is a crude estimate of the space used to store and manipulate the constraints. Phase I is the mono-variant analysis. Phase II is the poly-variant analysis (which uses information from phase I).

program		time(secs)	equations	space(MB)
intmap (105 lines)	phase I	0.35	1360	0.08
	phase II	0.37	1580	0.35
life (150 lines)	phase I	0.86	1925	0.18
	phase II	3.0	13769	1.7
lexgen (1170 lines)	phase I	3.4	6504	1.6
	phase II	3.7	9181	1.3

We expect substantial improvement in the running time and space of poly-variant analysis as the control of constraint duplication in further developed. Note that the space requirements of the poly-variant analysis are sometimes less than those for the mono-variant analysis. This is because the mono-variant analysis effectively folds different uses of a function together, and although this results in fewer set variables, it may substantially increase the number of constraints per variable (in the final explicit form constraints).

6 Conclusion

Starting with the simple intuition of treating program variables as sets, we have developed a powerful, general and flexible analysis for higher-order call-by-value functional languages. The contributions of the paper lie in two areas. First, in the very direct and appealing connection between a program's set based approximation (which is what our algorithm computes), and its

[10] The implementation collects all constraints with the same left-hand-side variable together, and the resulting object is effectively and equation.

underlying operational semantics. Second, in the presentation of an algorithm (and implementation thereof) which combines (a) an accurate treatment of data-structures, (b) modeling of side-effecting operations and (c) practicality.

Acknowledgments

Thanks to Olivier Danvy, Bob Harper, Peter Lee, Karoline Malmkjaer and David Tarditi for many useful discussions and comments at various stages of this work.

References

[1] A. Aiken and E. Wimmers, "Solving Systems of Set Constraints", *Proc. 7th IEEE Symp. on Logic in Computer Science*, Santa Cruz, pp. 329–340, June 1992.

[2] A. Aiken and E. Wimmers, "Type Inclusion and Type Inference", draft manuscript, January 1993.

[3] A. Appel, "Compiling with Continuations", Cambridge University Press, 1992.

[4] L. Bachmair, H. Ganzinger and U. Waldmann, "Set Constraints are the Monadic Class", Technical Report MPI–I–92–240, Max-Planck-Institute for Computer Science, December 1992.

[5] C. Consel and O. Danvy, "Tutorial Notes on Partial Evaluation", *Proc. 20th ACM Symp. on Principles of Programming Languages*, Charleston, pp. 493-501, January 1993.

[6] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", *Proc. 4th ACM Symp. on Principles of Programming Languages*, Los Angeles, pp. 238–252, January 1977.

[7] N. Heintze, "Set Based Program Analysis", Ph.D. thesis, School of Computer Science, Carnegie Mellon University, October 1992.

[8] N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs", *Proc. 17th ACM Symp. on Principles of Programming Languages*, San Francisco, pp. 197–209, January 1990. (A full version of this paper appears as IBM Technical Report RC 16089 (# 71415), 66 pp., August 1990.)

[9] N. Heintze and J. Jaffar, "A Decision Procedure for a Class of Herbrand Set Constraints", *Proc. 5th IEEE Symp. on Logic in Computer Science*, Philadelphia, pp. 42–51, June 1990. (A full version of this paper appears as Carnegie Mellon University Technical Report CMU–CS–91–110, 42 pp., February 1991.)

[10] N. Heintze and J. Jaffar, "An Engine for Logic Program Analysis", *Proc. 7th IEEE Symp. on Logic in Computer Science*, Santa Cruz, pp. 318–328, June 1992.

[11] F. Henglein, "Efficient Type Inference for Higher-Order Binding-Time Analysis", *Proceedings 5th ACM-FPCA*, Cambridge MA, LNCS 523, pp. 448-472, August 1991.

[12] N. Jones, "Flow Analysis of Lazy Higher-Order Functional Programs", in *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin (Eds.), Ellis Horwood, 1987.

[13] N. Jones, C. Gomard and P. Sestoft, "Partial Evaluation and Automatic Program Generation", "Prentice-Hall International", 1993.

[14] N. Jones and S. Muchnick, "Flow Analysis and Optimization of LISP-like Structures", *Proc. 6th ACM Symp. on Principles of Programming Languages*, San Antonio, pp. 244–256, January 1979.

[15] R. Milner, M. Tofte and R. Harper, "The Definition of Standard ML", MIT Press, 1990.

[16] P. Mishra and U. Reddy, "Declaration-free Type Checking", *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 7–21, January 1985.

[17] F. Nielson and H. Nielson, "Two-Level Functional Languages", Cambridge University Press, Vol 34, Cambridge Tracts in Theoretical Computer Science, 1992.

[18] J. Palsberg and M. Schwartzbach, "Safety Analysis versus Type Inference for Partial Types" Information Processing Letters, Vol 43, pp. 175-180, North-Holland, September 1992.

[19] J. Reynolds, "Automatic Computation of Data Set Definitions", *Information Processing 68*, pp. 456–461, North-Holland, 1969.

Appendix 1 : Construction of Set Constraints

$$\mathcal{Z} \rhd x : (\mathcal{X}_x, \{\}) \qquad \text{(VAR)}$$

$$\frac{\mathcal{Z} \rhd e_1 : (\mathcal{X}_1, C_1) \qquad \mathcal{Z} \rhd e_2 : (\mathcal{X}_2, C_2)}{\mathcal{Z} \rhd e_1\, e_2 : (\mathcal{Y}, \{\mathcal{Y} \supseteq \mathit{apply}(\mathcal{Y}', \mathcal{X}_2),\ \mathcal{Y}' \supseteq \mathit{ifnonempty}(\mathcal{Z}, \mathcal{X}_1)\} \cup C_1 \cup C_2)} \qquad \text{(APP)}$$

$$\frac{\mathcal{Z} \rhd e_i : (\mathcal{X}_i, C_i),\ i = 1..n}{\mathcal{Z} \rhd c(e_1, \ldots, e_n) : (\mathcal{Y}, \{\mathcal{Y} \supseteq c(\mathcal{X}_1, \ldots, \mathcal{X}_n)\} \cup C_1 \cup \ldots \cup C_n)} \qquad \text{(CONST)}$$

$$\frac{\mathcal{X}_x \rhd e : (\mathcal{X}, C)}{\mathcal{Z} \rhd \lambda x.e : (\mathcal{Y}, \{\mathcal{Y} \supseteq \lambda x.e,\ \mathit{ran}(\lambda x.e) \supseteq \mathcal{X}\} \cup C)} \qquad \text{(ABS)}$$

$$\frac{\mathcal{Z} \rhd e_1 : (\mathcal{X}_1, C_1) \qquad \mathcal{Z} \rhd e_2 : (\mathcal{X}_2, C_2) \qquad \mathcal{Z} \rhd e_3 : (\mathcal{X}_3, C_3)}{\mathcal{Z} \rhd \mathit{case}(e_1, c(x_1, \ldots, x_n) \Rightarrow e_2, y \Rightarrow e_3) : (\mathcal{Y}, C \cup C_1 \cup C_2 \cup C_3)} \qquad \text{(CASE)}$$

where $C = \{\mathcal{Y} \supseteq \mathit{case}(\mathcal{Y}', c(\mathcal{X}_{x_1}, \ldots, \mathcal{X}_{x_n}) \Rightarrow \mathcal{Z}_2, \mathcal{X}_y \Rightarrow \mathcal{Z}_3),\ \mathcal{Y}' \supseteq \mathit{ifnonempty}(\mathcal{Z}, \mathcal{X}_1)\}$

$$\frac{\mathcal{Z} \rhd e : (\mathcal{X}, C)}{\mathcal{Z} \rhd \mathit{fix}\, x.e : (\mathcal{X}_x, \{\mathcal{X}_x \supseteq \mathit{ifnonempty}(\mathcal{Z}, \mathcal{X})\} \cup C)} \qquad \text{(FIX)}$$

Figure 5: Construction of Set Constraints (complete version)

Figure 5 presents the complete details of the constructions of set constraints for a term. The main difference between Figure 5 and Figure 3 is that the relation $\mathcal{Z} \rhd e : (se, C)$ recursively passes down a set variable which is empty if the expression under consideration is never called, and is non-empty otherwise. The key property of the relation $\mathcal{Z} \rhd e : (se, C)$ is that if $\mathcal{Z}$ is empty then C is vacuously true, and if $\mathcal{Z}$ is nonempty, then se and C are equivalent to those constructed using the simpler deductive system in Figure 3. We now define $\mathcal{SC}(e_0)$ as follows: if $\mathcal{Z}$ is a new set variable and $\mathcal{Z} \rhd e : (\mathcal{X}, C)$, then $\mathcal{SC}(e_0)$ is the pair $(\mathcal{X}, \{\mathcal{Z} \supseteq e\} \cup C)$ where e is some arbitrary sc-value. Note that all sc-values are set expressions and that the choice of e is arbitrary – its only purpose is to force the variable $\mathcal{Z}$ to be nonempty, since otherwise the constraints C would be vacuously true.